

A Comparison of Three Programming Models for Adaptive Applications

Hongzhang Shan, Jaswinder Pal Singh, Leonid Oliker¹, Rupak Biswas²

Department of Computer Science, Princeton University, {shz, jps}@cs.princeton.edu

¹NERSC, MS 50F, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, loliker@lbl.gov

²Computer Science Corporation, MS T27A-1, NASA Ames research Center, Moffett Field, CA 94035,

rbiswas@nas.nasa.gov

Abstract

We study the performance and programming effort for two major classes of adaptive applications under three leading parallel programming models. We find that all three models can achieve scalable performance on the state-of-the-art multiprocessor machines. The basic parallel algorithms needed for different programming models to deliver their best performance are similar, but the implementations differ greatly, far beyond the fact of using explicit messages versus implicit loads/stores. Compared with MPI and SHMEM, CC-SAS (cache-coherent shared address space) provides substantial ease of programming at the conceptual and program orchestration level, which often leads to the performance gain. However it may also suffer from the poor spatial locality of physically distributed shared data on large number of processors. Our CC-SAS implementation of the PARMETIS partitioner itself runs faster than in the other two programming models, and generates more balanced result for our application.

1 Introduction

Architectural convergence has made it common for different programming models to be supported on the same platform. The three common programming models are MPI, SHMEM and CC-SAS. How these three programming models compare in terms of performance and ease of programmability is not clear. Our previously study [7, 8] has shown that for regular applications, using different programming models for the same application greatly affects performance as well as programming effort. In this paper, we will focus on adaptive applications in which the computational domain adapts to the evolution of the problem with time. Such applications require dynamic load balancing and exhibit inherent irregular and unpredictable access and communication patterns. They have become increasingly important in scientific and engineering fields as more complexed phenomena and domains are studied.

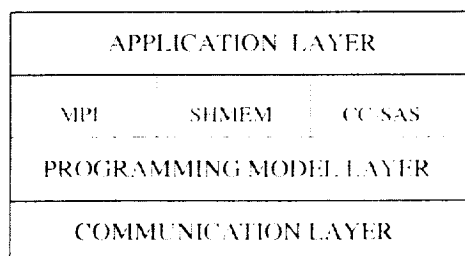


Figure 1: The layered platform for comparison of programming models

Our overall research goal is to study this problem in a layered framework that is shown in Figure 1. For the *application layer*, the applications selected should satisfy the following criteria: i)

They should require dynamic, irregular fine-grain communication. ii) They should have wide applicability to problem domains that require high-performance computing. iii) They should require the use of large numbers of processors. iv) They should not be trivial to obtain scalable performance. Based on these criteria, we select two typical applications *Adaptive Mesh* and *N-body* as our driving applications. For each application, one or more programs for each programming model is developed initially using leading known algorithms. For the *programming model layer*, the two dominant parallel programming paradigms are *message passing* and a *coherent shared address space*. There exists another programming model called *SHMEM* which lies in between this two. In this model, each process also has its own address space as in MPI, but the address spaces are symmetric. The communication becomes one-sided but still explicit. We will examine these three programming models in our study. For the *communication layer*, here, we focus on the tightly-coupled DSM multiprocessors. The platform we selected is the SGI Origin 2000, which has an aggressive communication architecture and provides full hardware support for CC-SAS model. The MPI and SHMEM programming models are built in software but leverage the hardware support for a shared address space and efficient communication for both ease of implementation and performance, as is increasingly the case in high-end tightly-coupled multiprocessors.

In this abstract, our main focus is on the application layer. There are two levels of consideration: algorithmic level and implementation level. We will examine whether the algorithms to deliver the best performance for each programming model are similar or not. If so, are there substantial differences in implementation level beyond just the fact of using explicit messages versus implicit loads and stores, and how much performance difference can be caused by them? How does the conceptual and programming complexity needed by different programming models for good performance compare? Lower-level considerations for the programming model layer and communication layer are left for the full paper.

Several researchers have done some previous work related with this problem. Singh et al find CC-SAS to have substantial ease of programming and, likely, performance advantages for hierarchical N-body applications [9, 10] on machines like the Stanford DASH. Dikaiakos and Stadel [1] studied the performance comparison of cosmological simulations between message passing and shared memory implementation on Intel Paragon and KSR-2 machines and found that the shared memory program running on KSR-2 outperforms the message passing program running on Intel Paragon. These platforms have become quite dated. Oliker and Biswas [5] examined the performance of a dynamic unstructured application on three different programming models. However each programming model was implemented on a different platform and performance can not be easily compared across them. These studies do not compare algorithmic and programming implications and their conclusions are different from ours which uses a common high performance platform with state-of-the-art implementation of the programming models. We find that all three models can achieve scalable performance on our platform. The algorithms needed by different programming models for best performance are similar,

but the implementations differs greatly, far beyond the fact of using explicit messages versus implicit loads/stores. They are at the conceptual and program orchestration levels. Compared with MPI or SHMEM, CC-SAS provides substantial ease of programming, which often leads to the performance gain. But it may also suffer from the poor spatial locality of physically distributed shared data on larger number of processors.

The rest of this abstract is organized as follows. Section 2 describes the applications we used and the programming differences for them among the three models. Performance is analyzed in Section 3. Finally, Section 4 summarizes our key conclusions.

2 Applications

In this section, we describe the algorithms for *Adaptive Mesh* and *N-body*. Since SHMEM programs are similar to MPI programs except for the two-sided versus one-sided communication, we only discuss the algorithmic differences between CC-SAS and MPI in this extended abstract. The complete discussion is left for the full paper. The discussion is focused on parallel partitioning for data locality and load balance.

2.1 Adaptive Mesh

The mesh used in our experiment is the one often used to simulate flow over an airfoil [5]. The flow-chart of the program is shown in Figure 2. The initial mesh is partitioned and each process is assigned a sub-mesh. Based on the sub-mesh, a corresponding sub-matrix is generated by each process and fed into the solver together. After the solver, the mesh adaptor will mark the edges and coarsen the mesh first. The refinement is delayed until after the load balancer which is responsible for re-partitioning the mesh and remapping the data. The following discussion highlights the stages where there are and aren't substantial differences in program orchestration due to the nature of the programming models, beyond just using messages instead of loads and stores, even though the same basic partitioning algorithms for load balance and communication reduction are used.

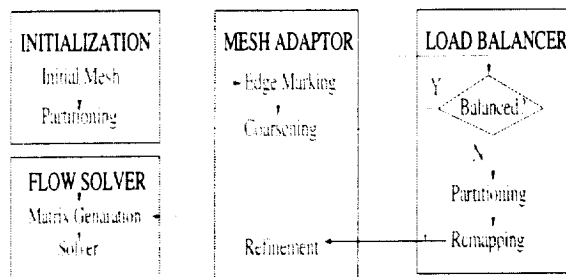


Figure 2: The flowchart of the adaptive mesh

Load Balancer In MPI programs, we chose PARMETIS [3] as the partitioner because of its good performance and availability. This is a multilevel partitioner, including coarsening, initial partitioning, and refining. The coarsening is implemented by heaviest-edge matching. To find matching vertices for the boundary vertices of a partition, a "try-confirm" stage is needed because if the matching vertex belongs to another process, the process has to send a message to another process to check whether that vertex has been matched by others or not. Then the sub-mesh has to be assembled for the initial partitioning. During the refining stage, each process will reconsider the ownership of its boundary vertices in order to reduce edge-cut and further balance the load. Due to the lack of globally shared data, the decision is made according to an inaccurate local view since address spaces are private. After the partitioning, the data is re-mapped among the processes. Each process has to break up its old sub-mesh and rebuild the new-sub-mesh. This remapping is very expensive for larger data sets.

In CC-SAS programs, there is only one complete mesh maintained by all processes. The "try-confirm" process is no longer needed and the communication to check for match-ability is replaced by synchronization. If a process finds that the matching vertex for vertex i is vertex j , it will lock vertex j , and check whether j has been already matched by other vertex or not. This is much easier to implement. Also, there is no need to assemble the sub-meshes before initial partitioning. In the refinement stage, when updating the ownership for border vertices, the shared address space enables a decision to be made based on the accurate global view, which helps to generate a more balanced partition. There is no explicit data remapping process necessary for program orchestration for CC-SAS.

Flow Solver Matrix generation is application dependent. We do not discuss it in this extended abstract.

For the solver, we use the publicly available Aztec [2] solver, which provides the state-of-the-art iterative methods for solving $Ax = b$. The matrix A is partitioned by rows among the processes. Each vertex in the mesh has a corresponding row in the matrix A . The partition comes from the load balancer.

The solver has been separated into two phases: matrix transform and iterative solver. In the transform stage, each process reorders its sub-matrix into a nearly block diagonal matrix to obtain good data locality for the time-consuming iterative solver phase. In MPI programs, this involves expensive hashing, searching and broadcasting operations due to all the data lying in private address spaces. In CC-SAS programs, a shared array is used to provide all the information needed by the reordering. Compared with MPI, the conceptual/orchestration complexity and programming effort is greatly reduced. The kernel of the iterative solver is a sparse matrix-vector multiply, which is similar across programming models except the differences of explicit message versus implicit loads/stores.

Mesh Adaptor In the mesh adaptor, all the edges are marked first to indicate whether they need to be bisected or collapsed based on the geometric information or error tolerance. The coarsening is done immediately after the marking. However, mesh refinement is delayed until after the load balancer. This delay will i)improve the load balance since the refinement is based on the new partition, ii)increase the data locality since the solver will work on the refined mesh, and iii)reduce the communication volume needed for data remapping after the repartitioning.

In CC-SAS programs, a complete shared mesh is maintained. A potential drawback is that the shared data structures can not be easily changed without synchronization. However, the need for synchronization can be dramatically reduced, often eliminated by letting each process compute its number of vertices (edges, elements) and apply the range to the global data structures in advance. This enables it to usually modify its partition of the data structures without synchronization. This increases complexity a little for the CC-SAS programs. However, the MPI programs have to maintain a lot of extra data structures to track data ownership and orchestrate communication. Combining all the components of this application together, the CC-SAS model provides substantial ease of programming despite the use of similar underlying partitioning algorithms.

2.2 N-body

In CC-SAS, there is a single copy of the global octree, which is built by having processors load their bodies concurrently into it, using locks to synchronize as necessary. Each processor is responsible for those bodies which were assigned to it in the force calculation phase in the last time step. The bodies are then partitioned using the *costzones* partitioning technique [9] so that every process has a contiguous of bodies range or zone with equal cost. Which *costzone* a body belongs to is determined by the total cost up to that body in an in-order traversal of the tree. The nodes of the tree are efficiently ordered in Peano-Hilbert order as a result. This ordering assures that the contiguity in the tree always corresponds to contiguity in space and therefore achieves good data locality.

In the MPI program, the shared tree is no longer available. Instead, each process builds a *locally essential tree*, which includes

all the body/node data that a process will need in the later force calculation stage. Unlike previous research that builds locally essential tree using an Orthogonal Recursive Bisection (ORB) partitioning [6, 4], we use a new method to base it on a partitioning scheme that is closer to the contiguous scheme used in the CC-SAS program for direct comparability. We also implement the ORB version and found it to have similar performance and programming complexities. Before building the tree, each process has to decide which bodies belong to it. First, the whole domain is partitioned into several sub-domains that are assigned to processes. Each process computes the cost distribution information for the bodies in its currently assigned subdomains. If the cost of a subdomain exceeds a threshold this subdomain will be further subdivided. In this way, we build a limited global tree (to a small number of levels) to represent the cost distribution. Next, by browsing this tree, a process can choose its subdomains according to a costzones like method and collect bodies falling into these subdomains from all other processes. The partitioning result should be similar to that of the CC-SAS program, though not exactly since it partitions subdomain cells rather than individual bodies. After collecting its bodies, a process first builds a local tree. Then it computes the body/node data from its local tree that is needed by every other processes, based on the subdomain partition information, and sends them to their destinations. After a process receives all the body/node data from other processes, it will add them into its local tree to generate the locally essential tree.

We can see that building the locally essential tree is much more complex conceptually and in orchestration than building the single shared tree in CC-SAS, although they are based on a very similar partitioning algorithm.

Finally we list the essential source code lines for all three models in Table 1. SHMEM is very similar to MPI because the main difference between them is the two-sided versus one-sided communication. CC-SAS needs far fewer lines than MPI or SHMEM. The difference is mainly caused by the substantial ease of programming provided by CC-SAS programming model at the conceptual and orchestration level.

	Adaptive Mesh				N-body
	Load Balancer	Flow Solver	Mesh Adaptor	Total	
MPI	5337	4615	6063	16015	1371
SHMEM	5579	4100	5906	15585	1322
CC-SAS	2563	2142	3725	8430	1065

Table 1: the number of essential source code lines

3 Performance

In this section, we will analyze performance using speedups relative to the same best sequential program in all cases (for each application) and per-process time breakdowns. The wall-clock time has been divided into four parts in the time breakdowns: BUSY (CPU time spent for computation), LMEM (CPU time waiting for local cache miss), RMEM (CPU time waiting for remote cache miss) and SYNC time (CPU time for synchronization). In CC-SAS programs, we could not differentiate the LMEM time and RMEM time using the available tools. We lump them together as MEM time.

3.1 Adaptive Mesh

The speedups of the entire adaptive mesh application are shown in Figure 3. The data set size means the number of triangles in the mesh. Figure 3 shows that all three models perform quite similarly for smaller data sets. With the increase of data set size, CC-SAS performs much better than the other two programming models.

The per-processor time breakdown for the largest (1.3M) data set size is shown in Figure 4. CC-SAS has much lower BUSY time compared with MPI and SHMEM. This is because i) the hashing, searching and broadcasting used in the transformation stage in Aztec for MPI or SHMEM are very expensive compared with the simpler implementation of CC-SAS (using one global array). ii) in

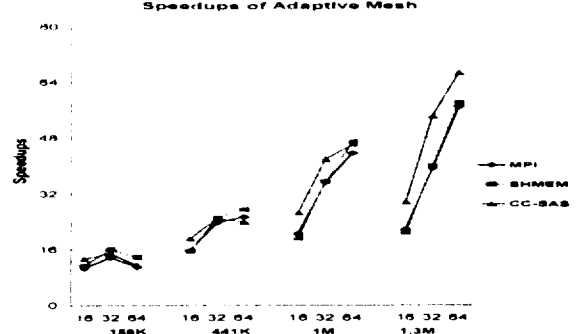


Figure 3: Speedups of adaptive mesh on 16, 32, and 64 processors for different data set sizes

the data remapping stage after the partitioning, the CC-SAS program does not need to break up and re-build the new sub-mesh as in MPI and SHMEM, since all the processes share the single copy of the complete mesh, and iii) in the load balancer, CC-SAS program does not need the "try-confirm" in the coarsening phase and the graph assembling in the initial partition phase. Among all these reasons, the transformation stage contributes most since the solver is the most time consuming part. Details of time and speedups for underlying phases will be included in the final paper. Thus, the programming advantages provided at the conceptual/orchestration level by the CC-SAS model directly lead to its better performance for larger data sets.

Interestingly, the information needed to make the transformation stage lower-overhead for MPI and SHMEM can be easily derived in the matrix generation phase. If this information is passed from matrix generator to the solver, the performance difference becomes much smaller. However this will hurt the solver's independence of its generation. One interesting issue is how to combine the matrix generation and flow solver together.

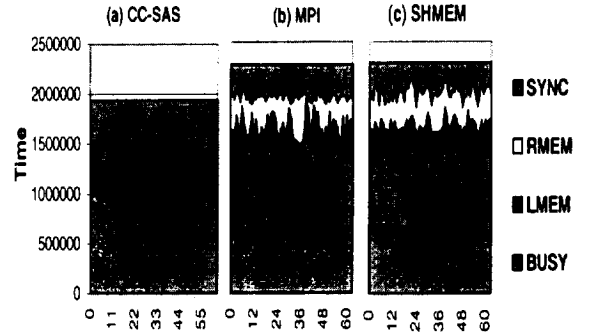


Figure 4: Time breakdown for 1.3M data set size on 64 processes

Compared with MPI or SHMEM, the CC-SAS version of the PARMETIS partitioner itself also works fast for our application due to its ease of programming discussed in Section 2. Figure 5 shows the wall-clock time needed to partition the graph we used (28404 vertices) for each programming model. The more processors, the more number of sub-meshes. Thus from 16 to 64 processors, the time is increased. CC-SAS partitioner also generates a more balanced result because in the refining stage, it can make a decision based on the accurate global view to update the vertex's ownership.

For adaptive mesh, all programming models can get good speedups (the parallel efficiency is above 80%). The basic parallel partitioning algorithms needed for the programming models are similar. CC-SAS has substantial ease of programming compared with MPI and SHMEM at the conceptual and orchestration level, which translates directly into its performance advantage.

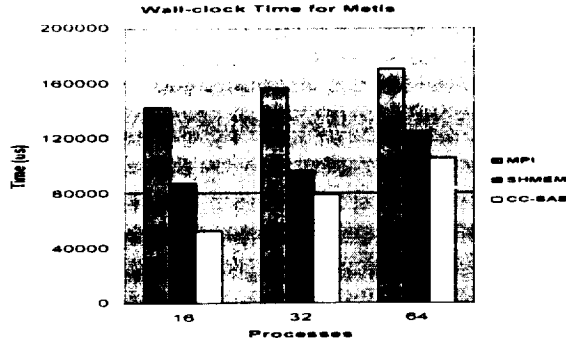


Figure 5: The wall-clock partitioning time for a graph with 28404 vertices on 16, 32 and 64 processors

3.2 N-body

The speedups of the N-body for these three models are shown in Figure 6. On 16 processors, they perform similarly across all the data sets. With the increase of number of processors, their behavior becomes different. For 16k bodies, CC-SAS performs best while for 1M bodies, CC-SAS falls far behind MPI and SHMEM.

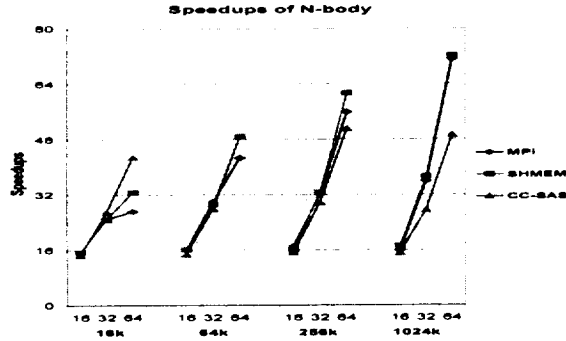


Figure 6: Speedups of N-body on 16, 32, and 64 processors for different data sets

To understand the performance difference, we show the time breakdowns for 16k (Figure 7) and 1 million (Figure 8) bodies on 64 processors. Figure 7 shows that the BUSY time for CC-SAS is much lower than that for MPI or SHMEM. This is because building the locally essential tree in MPI or SHMEM programs is much more complex and expensive than building a single shared tree in the CC-SAS program as we discussed in Section 2. The ease of programming provided by the CC-SAS programming model also brings performance advantages.

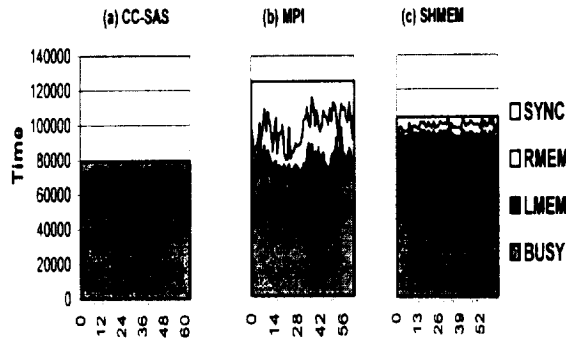


Figure 7: Time breakdown for 16k data set size on 64 processes

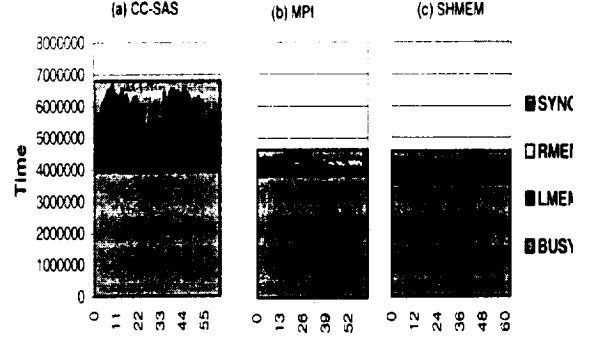


Figure 8: Time breakdown for 1024k data set size on 64 processes

The time breakdown for 1 million bodies is quite different. The BUSY time is close to each other for all three programming models. The extra overhead in the tree-building phase for MPI and SHMEM is no longer important since the total execution time even in the parallel case is dominated by force calculation. It's the MEM time in CC-SAS that is much higher and imbalanced (see Figure 8) this time. The reason is that during the force calculation phase, in MPI and SHMEM programs, the force calculation is completely local without any interprocess communication. However in CC-SAS, the shared tree is physically scattered among all the processors. These data do not have good spatial data locality. When a processor reference the data in the tree, those data are often allocated in other process's memory. This causes a lot of TLB misses and hurts the performance.

The problem could be improved by duplicating those remote cells in the tree locally, especially those cells closer to the root cell since they are most frequently referenced. The effect is similar to building a locally essential tree, but only into a limited levels.

4 Conclusion

The performance effects of programming models for adaptive applications are not consistent. We studied this problem in a layered approach and find that all three programming models (CC-SAS, SHMEM, MPI) can achieve scalable performance on the state-of-the-art multiprocessor machines. The fundamental parallel partitioning algorithms to deliver the best performance for each programming model are similar. However, the implementation differs greatly, even in conceptual ways, at the orchestration level, far beyond the fact of using explicit messages versus implicit loads and stores. CC-SAS provides substantial ease of programming, which often translates to the performance gain. However, the performance of CC-SAS may suffer from the poor spatial locality of the physically distributed shared data, in which case, some (but not all) of the ease of programming advantages must be given up to obtain comparable performance. The CC-SAS implementation of the well-known METIS partitioner works fast than PARMETIS for our applications. It also generates a more balanced partition because of the more accurate global view.

References

- [1] Marios Dikaiakos and Joachim Stadel. A performance study of cosmological simulations on message-passing and shared memory multiprocessors. In *ICS '96 Philadelphia, USA*, 1996.
- [2] <http://www.cs.sandia.gov/CRF/actec1.html>. Aztec: a massively parallel iterative solver library for solving sparse linear systems.
- [3] George Karypis. PARMETIS. <http://www-users.cs.umn.edu/karypis/metis>.
- [4] Pangfeng Liu and Sandeep N. Bhatt. experiences with parallel N-body simulation. In *SPAA 94, Cape May, NJ, USA*, June 1994.
- [5] Leonid Oliker and Rupak Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. In *Proceedings of SC99, Portland, 1999*.

- [6] John K. Salmon. *Parallel Hierarchical N-body Methods*. PhD thesis, California Institute of Technology, December 1990.
- [7] Hongzhang Shan and Jawinder Pal Singh. Comparison of message passing, SHMEM and cache-coherent shared address space programming models on the SGI Origin 2000. In *International Conference on Supercomputing*, June 1999.
- [8] Hongzhang Shan and Jawinder Pal Singh. Parallel sorting on cache-coherent DSM multiprocessors. In *Supercomputing*, November 1999.
- [9] Jaswinder Pal Singh, Anoop Gupta, and John L. Hennessy. Implications of hierarchical N-body techniques for multiprocessor architecture. *ACM Transactions on Computer Systems*, May 1995.
- [10] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: performance and architectural implications. *IEEE Computer*, 27(6), June 1994.